CS 344: Operating Systems I

03.01: Part IV – Synchronization

Mon/Wed 12:00 – 1:50 PM

Sanghyun Hong

sanghyun.hong@oregonstate.edu

Oregon State University

SAIL
Secure AI Systems Lab

Notice

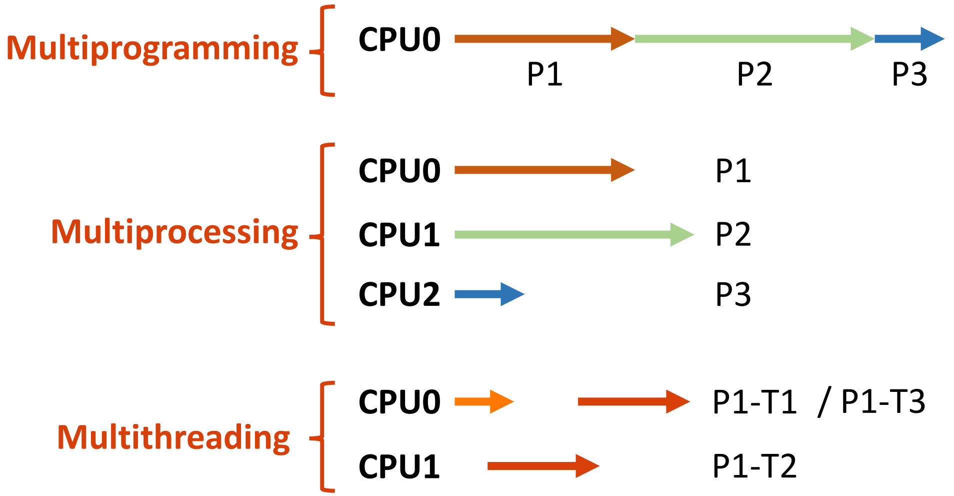
- Announcements
  - Extra credit opportunities on Canvas (12%)
    - Rust Programming Practice (+2%)
    - Build an ML classifier (+2%)
    - Multi-process data loader (+3%)
    - Some articles about Linus Torvalds (+5%)

Topics for today

- Part IV – Synchronization
  - Recap:
    - Terminology
    - Process (or thread) scheduling
  - Manage resources
    - Race condition (ATM server's problem)
  - Provide abstraction & Offer standard interface
    - Atomic operation
    - Mutual exclusion (mutex)

Recap: terminology

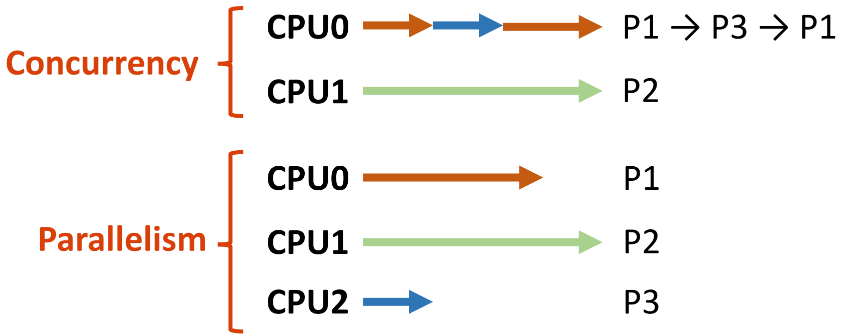
- **Three terms**
  - Multi-programming: multiple jobs running (or multiple programs in memory)
  - Multi-processing: multiple processors (multiple CPUs)
  - Multi-threading: multiple threads



Recap: terminology

- **Concurrency vs. parallelism:**
  - Concurrency: handling multiple processes (or threads) at once
  - Parallelism: running multiple processes (or threads) *simultaneously*

- **Example:**
  - On the CPU0
    - P1 and P3 can execute *concurrently*
    - P1 and P3 is *not* running in parallel
  - On the CPU0 and CPU1
    - P1 and P2 runs in parallel

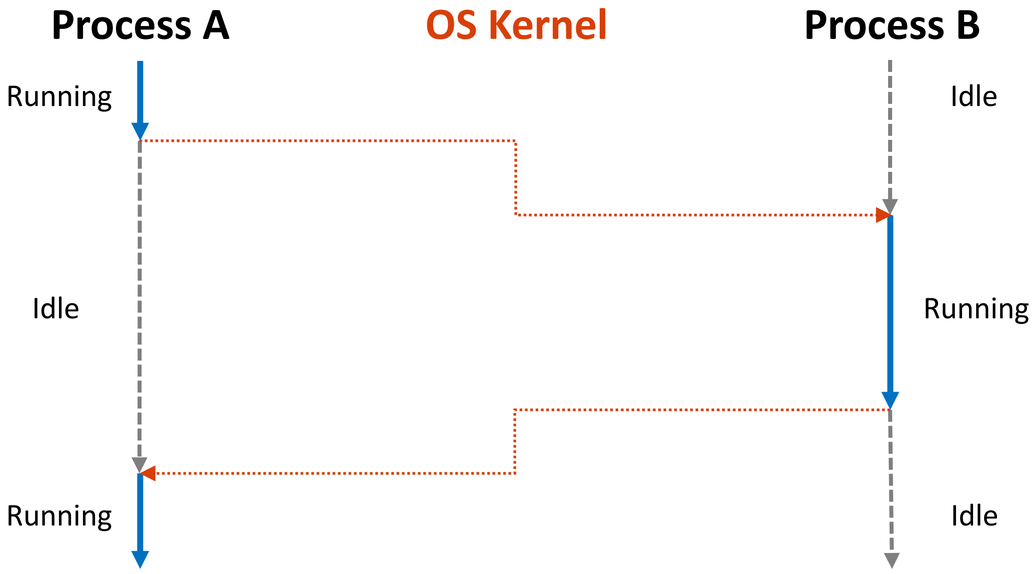


Recap: context switch

- **Context switch**
  - **Definition:** OS stores the current process's status and loads the new process's one
  - **Informal:** OS takes a CPU from one process and gives it to another

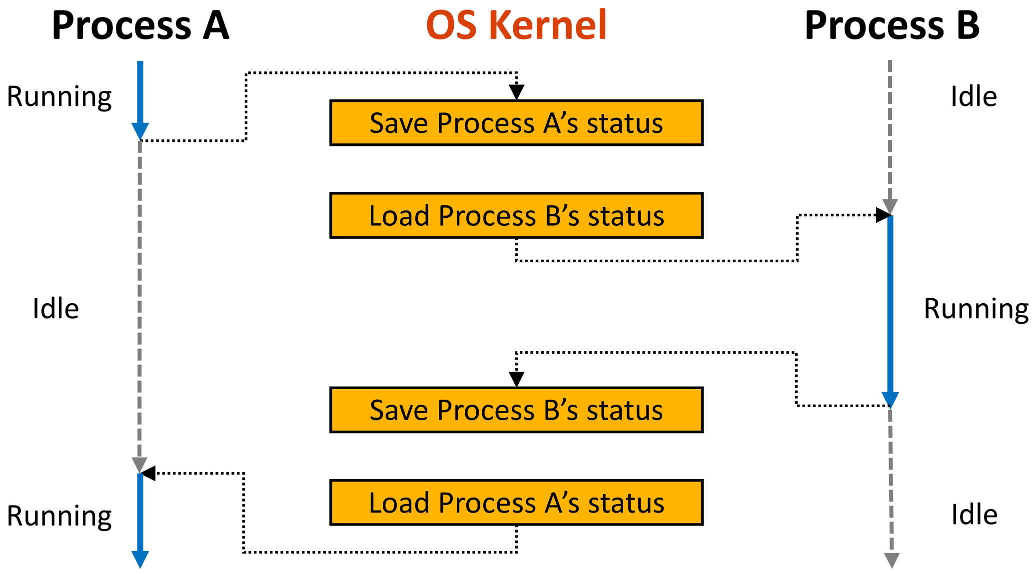
Recap: context switch – cont'd

- **Context switch**
  - **Definition:** OS stores the current process's status and loads the new process's one
  - **Informal:** OS takes a CPU from one process and gives it to another



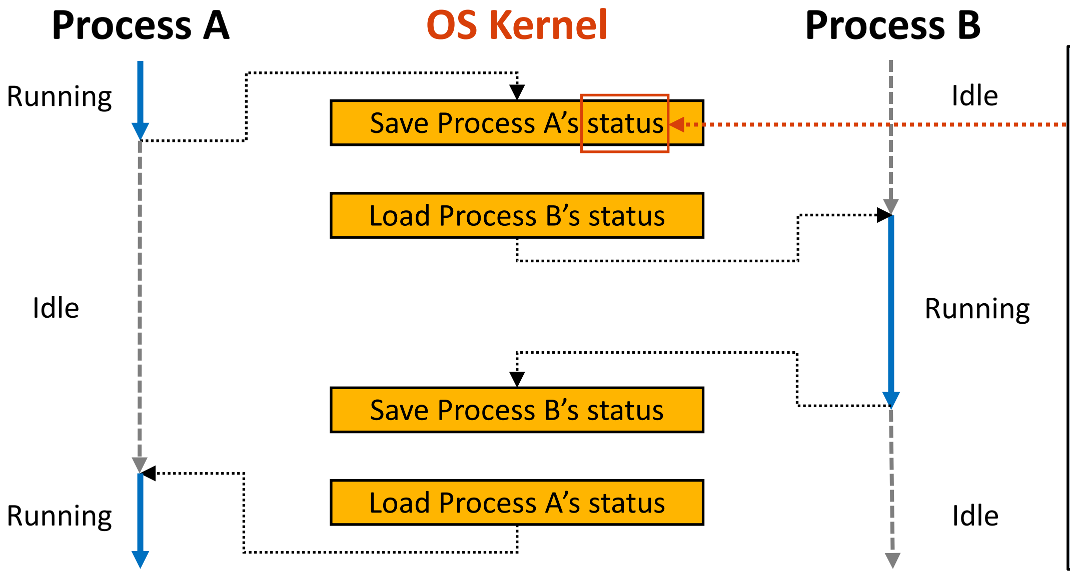
Recap: context switch – cont'd

- **Context switch**
  - **Definition:** OS stores the current process's status and loads the new process's one
  - **Informal:** OS takes a CPU from one process and gives it to another



Recap: Context switch – cont'd

- **Context switch**
  - **Definition:** OS stores the current process's status and loads the new process's one
  - **Informal:** OS takes a CPU from one process and gives it to another



Recall: Process control block

A structure in OS that contains a set of information required to run a process on a CPU. Recall that Linux has *task_struct*.

- CPU#
- Program counter
- Instruction pointer
- Heap/stack pointer
- Process state [!]
- ...

Recap: process context

- (Linux) has the process context
  - **Code**
    - Program counter
    - Instruction pointer
  - **Stack and heap**
    - Stack pointer
    - Heap pointer
  - **Running context**
    - Process state (ID, …)
    - Execution flags
    - CPU # to run
    - (OS II) Scheduling policy
    - (OS II) Mem. virtualization
  - …

Process Context: A set of information that OS requires to run a process on a CPU, different from CPU vendors (ex. In Linux, it's defined as *task_struct*, Link)

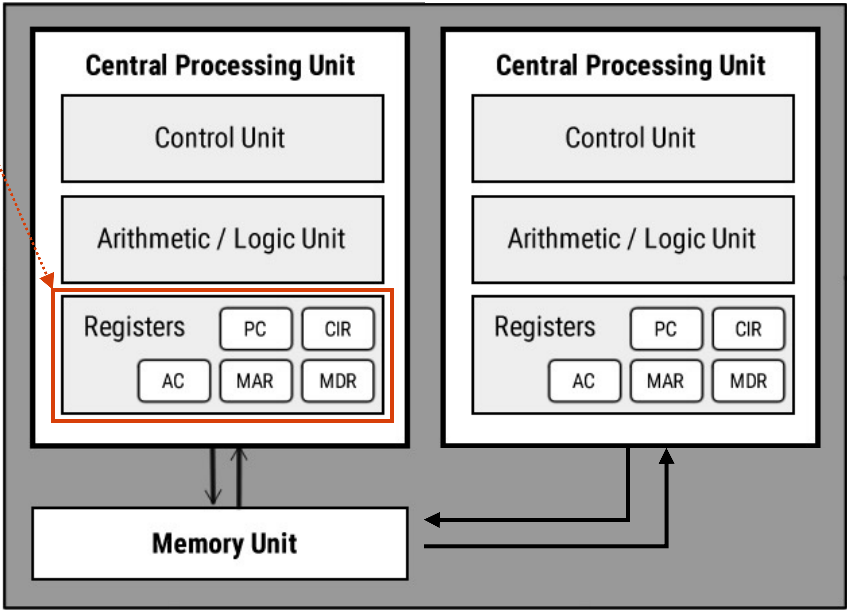
```
struct task_struct {
#ifdef CONFIG_THREAD_INFO_IN_TASK
	/*
	 * For reasons of header soup (see current_thread_info()), this
	 * must be the first element of task_struct.
	 */
	struct thread_info		thread_info;
#endif
	unsigned int			__state;

#ifdef CONFIG_PREEMPT_RT
	/* saved state for "spinlock sleepers" */
	unsigned int			saved_state;
#endif

	/*
	 * This begins the randomizable portion of task_struct. Only
	 * scheduling-critical items should be added above here.
	 */
	randomized_struct_fields_start

	void				*stack;
	refcount_t			usage;
	/* Per task flags (PF_*), defined further below: */
	unsigned int			flags;
	unsigned int			ptrace;
```

```
	struct sched_info		sched_info;

	struct list_head		tasks;
#ifdef CONFIG_SMP
	struct plist_node		pushable_tasks;
	struct rb_node			pushable_dl_tasks;
#endif

	struct mm_struct		*mm;
	struct mm_struct		*active_mm;

	/* Per-thread vma caching: */
	struct vmacache			vmacache;

#ifdef SPLIT_RSS_COUNTING
	struct task_rss_stat		rss_stat;
#endif
	int				exit_state;
	int				exit_code;
	int				exit_signal;
	/* The signal sent when the parent dies: */
	int				pdeath_signal;
	/* JOBCTL_*, siglock protected: */
	unsigned long			jobctl;

	/* Used for emulating ABI behavior of previous Linux versions: */
	unsigned int			personality;
```

Recap: process context – cont'd

- (Linux) has the process context
  - Code
    - Program counter
    - Instruction pointer
  - Stack and heap
    - Stack pointer
    - Heap pointer
  - Running context
    - Process state (ID, …)
    - Execution flags
    - CPU # to run
    - (OS II) Scheduling policy
    - (OS II) Mem. virtualization
  - …

Process Context: A set of information that OS requires to run a process on a CPU, different from CPU vendors (ex. In Linux, it's defined as *task_struct*, Link)

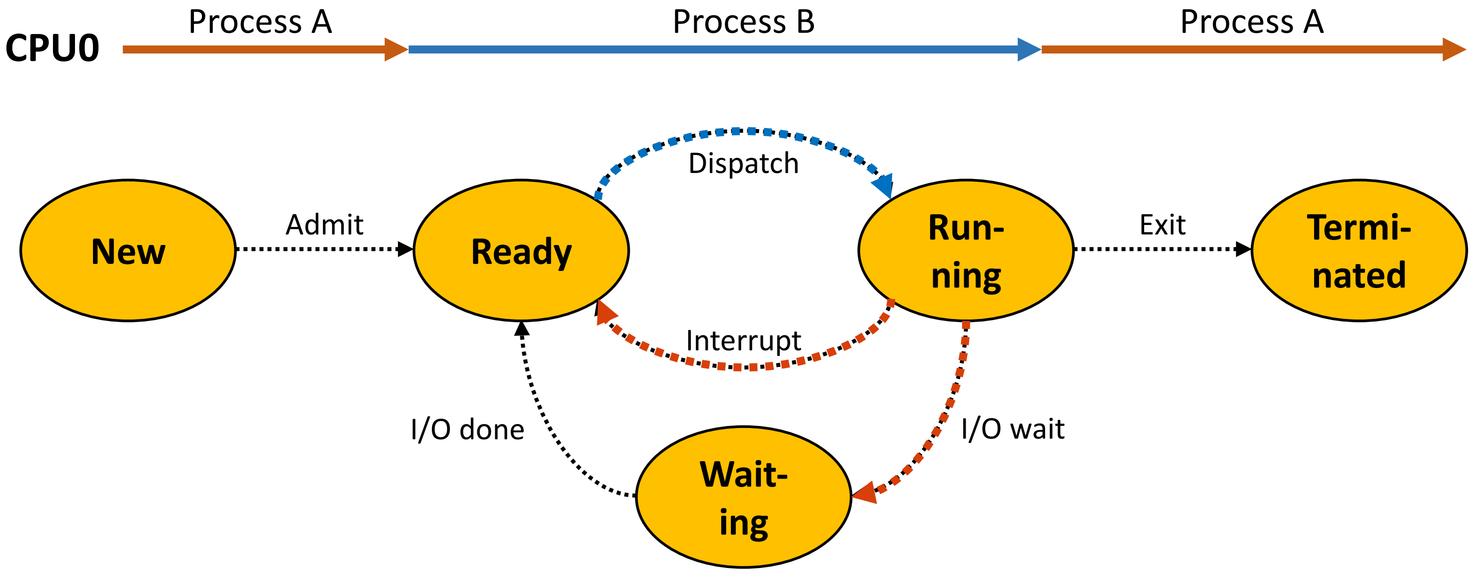


Recap: process state

- A process can have five states:
  - **New:** a process (or thread) is being created (by fork())
  - **Ready:** the process is waiting to run
  - **Running:** the process is running on a CPU(or CPUs)
  - **Waiting:** the process is waiting for some events to occur (*e.g.*, a data loaded from storage)
  - **Terminated:** the process has finished execution; waiting for removal

Recap: Process State Transition

- **Context switch**
  - **Definition:** OS stores the current process's status and loads the new process's one
  - **Informal:** OS takes a CPU from one process and gives it to another



Recap: process scheduling

- **Scheduling**
  - **Definition:** an OS activity that schedules processes in different states
  - **Note:** OS implements queues to hold multiple processes in the same state

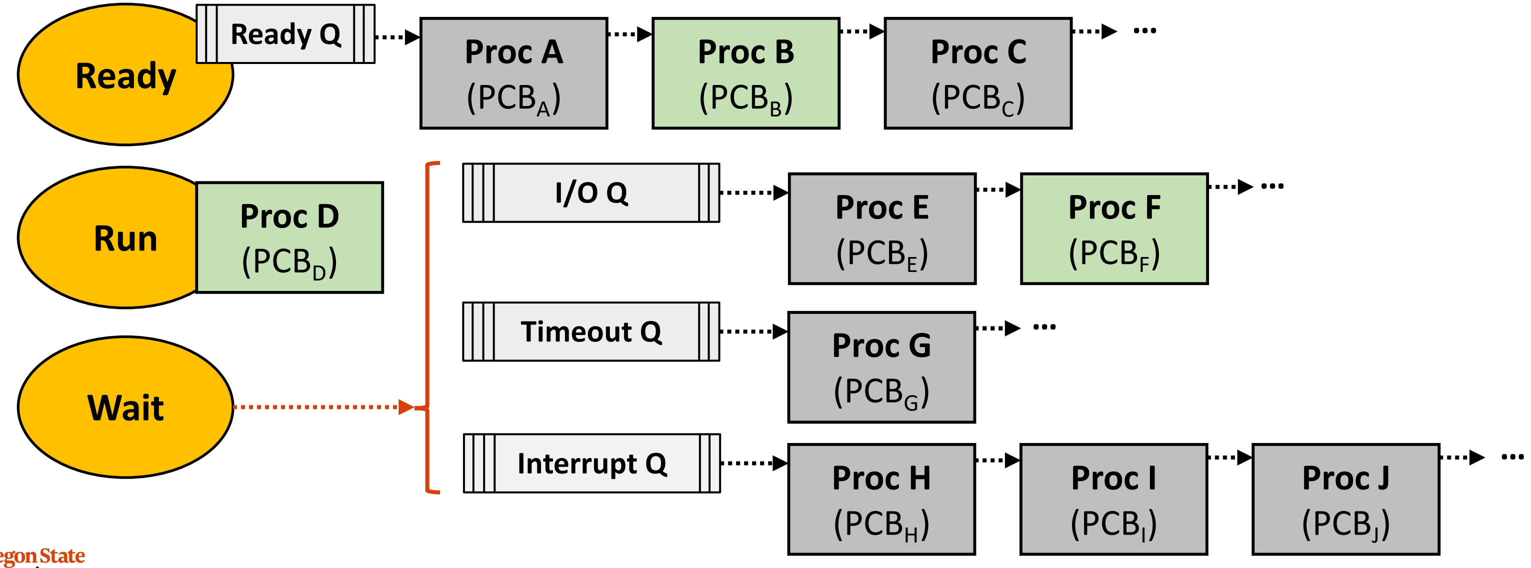
RECAP: PROCESS SCHEDULING

- **Scheduling**
  - **Definition:** an OS activity that schedules processes in differe
  - **Note:** OS implements queues to hold multiple processes in t

Illustrated Example

1. OS kicks out Proc D (timeout)
2. OS runs Proc B
3. OS puts Proc F in the ready Q
   (I/O has been done, in this case)

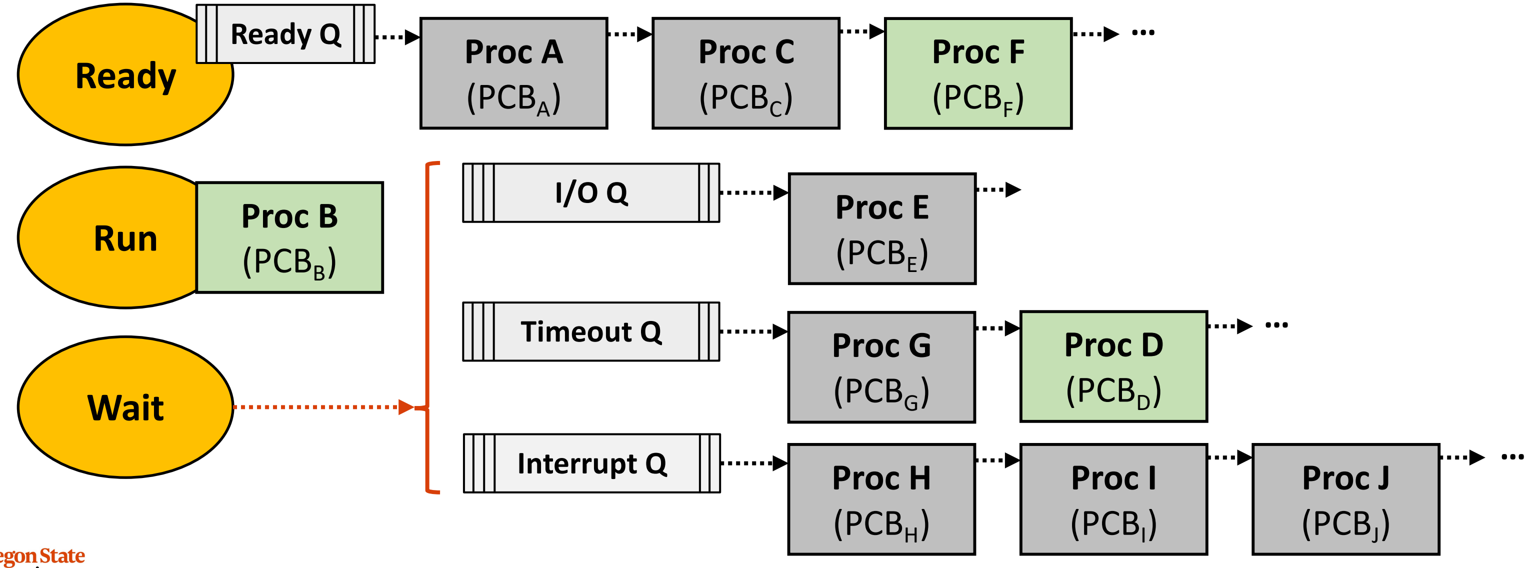
- **Illustration (single CPU)**



Recap: process queues

- **Process queues in Linux**
  - Separate queue for each kick-out conditions (I/O, timeout, etc…)
  - OS does **not** pick a PCB from each queue in a FIFO manner

- **Illustration (single CPU)**



Recap: OS scheduler

- **(OS) Scheduler:**
  - **Definition:** An OS task (process) that manages the process scheduling activity

- **Implementation**

```
while ( <some condition,
            but eventually will be infinite>) {

   RunProcess( curProc );
   newProc = chooseNextProc();
   saveCurrentProc( curProc );
   LoadNextState( newProc );

}
```

  - It is also a process (an *infinite* loop)
  - The scheduler process terminates if we *stop* (turn-off) a computer

OS SCHEDULER

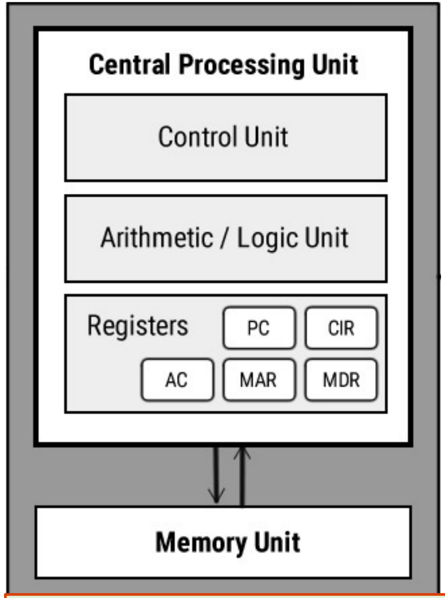
- **How OS scheduler works?**

```
while ( <some condition,
          but eventually will be infinite>) {

    RunProcess( curProc );
    newProc = chooseNextProc();
    saveCurrentProc( curProc );
    LoadNextState( newProc );

}
```

  - RunProcess(): a CPU executes the machine code of “curProc”

PCB$_{curProc}$

CPU#
Prog. counter
Heap/Stack



curProc (a = 5 + 8)

Program counter (PC) →

LDR	r5	8	// load 8
LDR	r4	5	// load 5
ADD	r5	r4	// add two
PUSH	r5	0x20	// store it

OS scheduler – cont'd

- **How OS scheduler works?**

```
while ( <some condition,
          but eventually will be infinite>) {

    RunProcess( curProc );
    newProc = chooseNextProc();
    saveCurrentProc( curProc );
    LoadNextState( newProc );

}
```

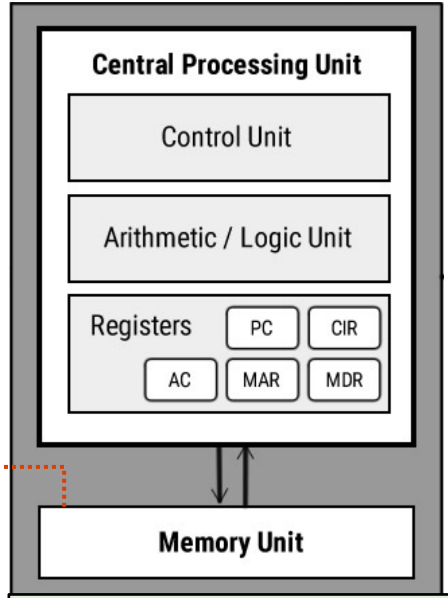
  - RunProcess(): a CPU executes the machine code of "curProc"
  - chooseNextProc(): OS kernel selects the next process to run

$PCB_{curProc}$

CPU#
Prog. counter
Heap/Stack

$PCB_{newProc}$

CPU#
Prog. counter
Heap/Stack



curProc (a = 5 + 8)

LDR	r5	8	// load 8
LDR	r4	5	// load 5
ADD	r5	r4	// add two
PUSH	r5	0x20	// store it

Program counter (PC) → PUSH r5 0x20

OS scheduler – cont'd

- **How OS scheduler works?**

```
while ( <some condition,
            but eventually will be infinite>) {

    RunProcess( curProc );
    newProc = chooseNextProc();
    saveCurrentProc( curProc );
    LoadNextState( newProc );

}
```

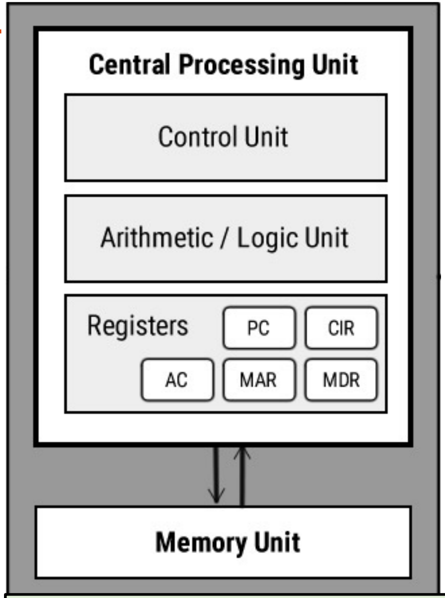
  – RunProcess(): a CPU executes the machine code of "curProc"
  – chooseNextProc(): OS kernel selects the next process to run
  – saveCurrentProc(): OS kernel saves the CPU's state to "curProc"

$PCB_{curProc}$

CPU#
Prog. counter
Heap/Stack

$PCB_{newProc}$

CPU#
Prog. counter
Heap/Stack



curProc (a = 5 + 8)

LDR	r5	8	// load 8
LDR	r4	5	// load 5
ADD	r5	r4	// add two
PUSH	r5	0x20	// store it

Program counter (PC) → PUSH

OS SCHEDULER – CONT'D

- **How OS scheduler works?**

```
while ( <some condition,
            but eventually will be infinite>) {

    RunProcess( curProc );
    newProc = chooseNextProc();
    saveCurrentProc( curProc );
    LoadNextState( newProc );

}
```

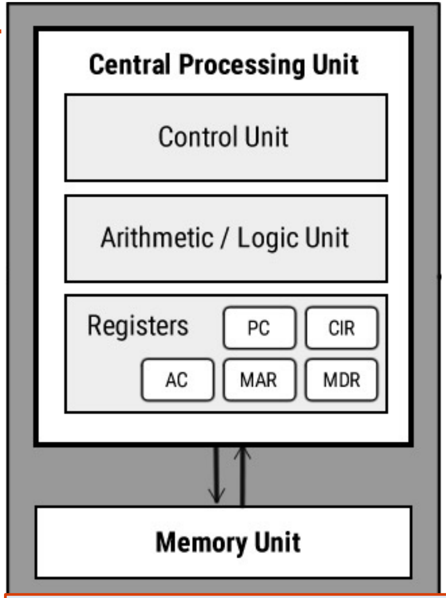
  - RunProcess(): a CPU executes the machine code of "curProc"
  - chooseNextProc(): OS kernel selects the next process to run
  - saveCurrentProc(): OS kernel saves the CPU's state to "curProc"
  - loadNextState(): OS kernel stores "newProc" state to the CPU

$PCB_{curProc}$

CPU#
Prog. counter
Heap/Stack

$PCB_{newProc}$

CPU#
Prog. counter
Heap/Stack



newProc (a = 8 - 5)

LDR	r5	8	// load 8
LDR	r4	5	// load 5
SUB	r5	r4	// sub two
PUSH	r5	0x20	// store it

Program counter (PC) → SUB

OS scheduler – cont'd

- **What triggers OS scheduling?**

```
while ( <some condition,
            but eventually will be infinite>) {

    RunProcess( curProc );
    newProc = chooseNextProc();   <······· Yield or interrupt triggers this code line
    saveCurrentProc( curProc );
    LoadNextState( newProc );

}
```

  - RunProcess(): a CPU executes the machine code of "curProc"
  - chooseNextProc(): OS kernel selects the next process to run
  - saveCurrentProc(): OS kernel saves the CPU's state to "curProc"
  - loadNextState(): OS kernel stores "newProc" state to the CPU

OS SCHEDULER: YIELD

- **Two mechanisms (that triggers** chooseNextProc()**)**
  - **Yield:** a process *voluntarily* gives a CPU away
  - **Interrupt:** an external event happens, and OS kernel *preemptively* runs it

- **Yield Example** (in your program)

```
void fn() {
    …

    // write the data in buf to an I/O device
    fwrite(str, sizeof(char), sizeof(buf), fp);
    sched_yield();  <-------- Your program pauses at here
                              Then, context switching happens
                              OS kernel schedules a new process on a CPU
}

int main(void) {
    fn();
}
```

OS Scheduler: Interrupt

- **Two mechanisms (that triggers** chooseNextProc()**)**
  - **Yield:** a process *voluntarily* gives a CPU away
  - **Interrupt:** an external event happens, and OS kernel *preemptively* runs it

- **Interrupt Example** (in your program)

```
void fn() {
    …

    // write the data in buf to an I/O device
    write(fd, buf, wlen);
    printf("Data is written: %d\n", wlen);

}

int main(void) {
    fn();
}
```

← Your program waits until the write operation finishes and while waiting, OS kernel schedules another process

OS SCHEDULER: INTERRUPT

- **Two mechanisms (that triggers** chooseNextProc()**)**
  - **Yield:** a process *voluntarily* gives a CPU away
  - **Interrupt:** an external event happens, and OS kernel *preemptively* runs it

- **Interrupt Example** (in your program)

```
void fn() {
    …

    // write the data in buf to an I/O device
    write(fd, buf, wlen);
    printf("Data is written: %d\n", wlen);

}

int main(void) {
    fn();
}
```

← Once the write is done, OS receives “done” from the disk OS then schedules your proc on a CPU and it runs this line

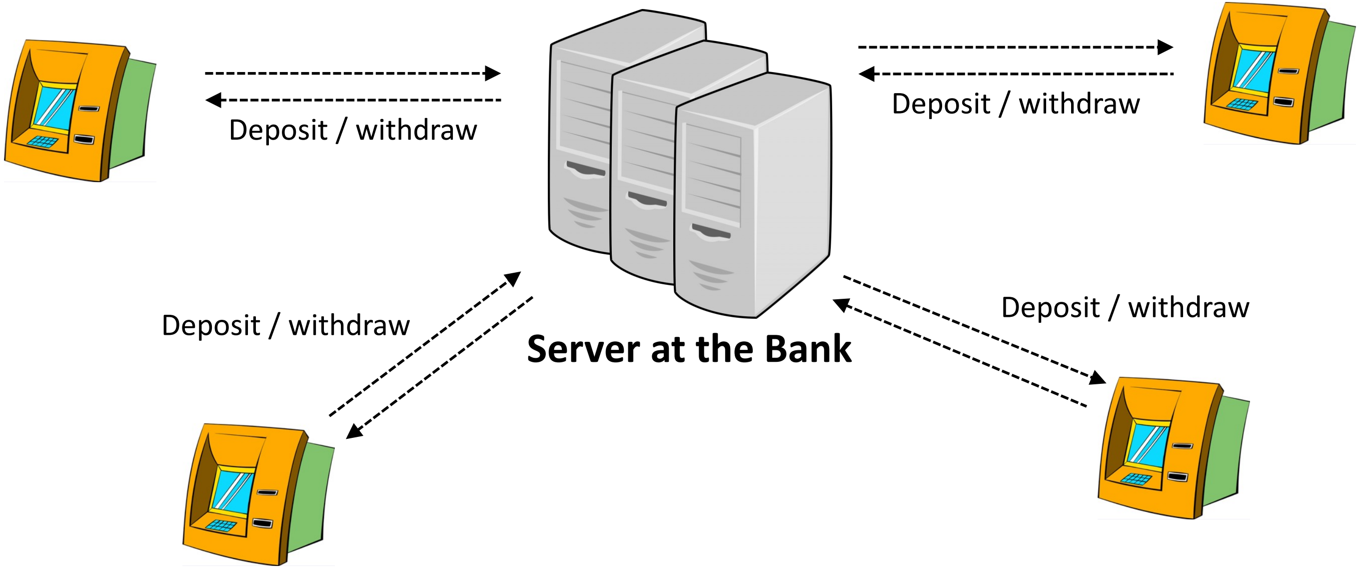
How It Works? Take CS 444: OS II

Topics for today

- Part IV – Synchronization
  - Recap:
    - Terminology
    - Process (or thread) scheduling
  - Manage resources
    - Race condition (ATM server’s problem)
  - Provide abstraction & Offer standard interface
    - Atomic operation
    - Mutual exclusion (mutex)

Synchronization

- **ATM bank's server**
  - The server(s) takes care of multiple deposit / withdrawal requests
  - Bank want to make sure all the transactions are correct



Synchronization: ATM bank server v0.1

- **Server in C**
  - Receive a request
  - Process the request
  - Perform those actions *iteratively*

- **Potential problem**
  - A single request at a time
  - Problem: ~470k ATMs in the US (2018)

```
void ProcessRequest(op, accountId, amount) {
   switch (op) {
      case OP_DEPOSIT:
         Deposit(accountId, amount);
      case OP_WITHDRAW:
         Withdraw(accountId, amount);
         … <here, you can define more ops…>
   }
}

void Deposit(accountId, amount) {
   account = GetAccount(accountId);
   account->balance += amount;
   StoreAccount(account);
}

int main(void) {
   int op = -1;
   int accountId = -1;
   int amount = -1;

   while (1) {
      ReceiveRequest(&op, &accountId, &amount);
      ProcessRequest(op, accountId, amount);
   }

   return 0;        // code only reaches here if the server terminates
}
```

Synchronization: ATM bank server v0.2

- **Event-driven ATM bank server**
  - Receive/process events
  - Store them to a buffer
  - Deposit when "account" is available

- **Potential problem:**
  - Increase implementation complexity
  - How many events do we need?

```
struct Event {
   int eventType;
   int accountId;
   int amount;
   struct account* account;
};

void PullAccount(struct Event* event) {
   event->account = GetAccount(event->accountId);
}

void Deposit(struct Event* event) {
   event->account->balance += event->amount;
   event->amount = 0;
}

int main(void) {
   …

   while (1) {
      event = Wait4NextEvent();
      if (event->eventType == RequestReceived)      PullAccount(event);
      else if (event->eventType == DepositReady)    Deposit(event);
   }

   return 0;        // code only reaches here if the server terminates
}
```

SYNCHRONIZATION: ATM BANK SERVER V0.3

- **Threaded ATM bank server**
  - Receive a request
  - Create a thread for processing it
  - Multiple threads can co-exist

- **Potential problem:**

Thread A	**Thread B**
1. Load my balance: $400	
	2. Load my balance: $400
	3. Deposit $100
4. Deposit $200	

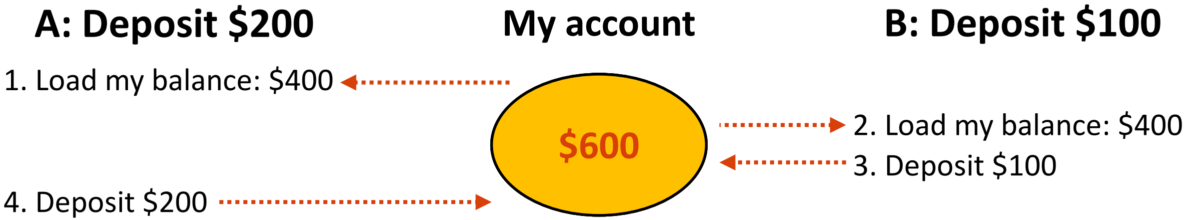
Now, What's My Balance?

```
void ProcessRequest(op, accountId, amount) {
   switch (op) {
      case OP_DEPOSIT:
         pthread_t *newTh = <mem alloc>;
         pthread_create(newTh, Deposit, info);
      case OP_WITHDRAW:
         pthread_t *newTh = <mem alloc>;
         pthread_create(newTh, Withdraw, info);
   }
}
```

```
void Deposit(accountId, amount) {
   account = GetAccount(accountId);
   account->balance += amount;
   StoreAccount(account);
}

int main(void) {
   int op = -1;
   int accountId, amount = -1, -1;

   while (1) {
      ReceiveRequest(&op, &accountId, &amount);
      ProcessRequest(op, accountId, amount);
   }

   return 0;          // code only reaches here if the server terminates
}
```

SYNCHRONIZATION: RACE CONDITION

- **Race condition:**
  - **Definition:** an undesirable scenario; performs multiple operations on **a shared resource**
  - **Example:** two "deposit" threads, running *concurrently*, increase the balance



How Can We Make Sure My Balance Is $700 at the End?

Topics for today

- Part IV – Synchronization
  - Recap:
    - Terminology
    - Process (or thread) scheduling
  - Manage resources
    - Race condition (ATM server's problem)
  - Provide abstraction & Offer standard interface
    - Atomic operation
    - Mutual exclusion (mutex)

Synchronization: atomic operation

- **Solution approach:**
  - Deposit() is not *indivisible*
  - Make sure to execute "Deposit()" at once

- **Atomic operation:**
  - Code should be executed w/o interrupt
  - **TL; DR:** Code should be run *at once*

```
void ProcessRequest(op, accountId, amount) {
  switch (op) {
    case OP_DEPOSIT:
      pthread_t *newTh = <mem alloc>;
      pthread_create(newTh, Deposit, info);
    case OP_WITHDRAW:
      pthread_t *newTh = <mem alloc>;
      pthread_create(newTh, Withdraw, info);
  }
}

void Deposit(accountId, amount) {
  account = GetAccount(accountId);
  account->balance += amount;
  StoreAccount(account);
}

int main(void) {
  int op = -1;
  int accountId, amount = -1, -1;

  while (1) {
    ReceiveRequest(&op, &accountId, &amount);
    ProcessRequest(op, accountId, amount);
  }

  return 0;          // code only reaches here if the server terminates
}
```

Synchronization: Mutual Exclusion (Mutex)

- Mutex (lock)
  - Prevents two+ process access the code
  - Supports three operations
    - Lock before running atomic code
    - Unlock after running the code
    - Wait while someone locked the code

```
pthread_mutex_t deposit_lock;

void ProcessRequest(op, accountId, amount) {
   switch (op) {
      case OP_DEPOSIT:
         …
   }
}

void Deposit(accountId, amount) {
   pthread_mutex_lock(&foo_mutex);         // lock before the atomic op.
   account = GetAccount(accountId);
   account->balance += amount;
   StoreAccount(account);
   pthread_mutex_unlock(&foo_mutex);       // unlock after the atomic op.
}

int main(void) {
   int op = -1;
   int accountId, amount = -1, -1;
   pthread_mutex_init(&deposit_lock, NULL);

   while (1) {
      ReceiveRequest(&op, &accountId, &amount);
      ProcessRequest(op, accountId, amount);
   }

   return 0;          // code only reaches here if the server terminates
```

SYNCHRONIZATION: MUTUAL EXCLUSION (MUTEX)

- Mutex (lock)
  - Prevents two+ process access the code
  - Supports three operations
    - Lock before running atomic code
    - Unlock after running the code
    - Wait while someone locked the code
- Critical section
  - A code section protected by lock & unlock

```
pthread_mutex_t deposit_lock;

void ProcessRequest(op, accountId, amount) {
   switch (op) {
      case OP_DEPOSIT:
         …
   }
}

void Deposit(accountId, amount) {
   pthread_mutex_lock(&foo_mutex);         // lock before the atomic op.
   account = GetAccount(accountId);
   account->balance += amount;
   StoreAccount(account);
   pthread_mutex_unlock(&foo_mutex);       // unlock after the atomic op.
}

int main(void) {
   int op = -1;
   int accountId, amount = -1, -1;
   pthread_mutex_init(&deposit_lock, NULL);

   while (1) {
      ReceiveRequest(&op, &accountId, &amount);
      ProcessRequest(op, accountId, amount);
   }

   return 0;         // code only reaches here if the server terminates
```

SYNCHRONIZATION: MUTUAL EXCLUSION (MUTEX)

- Mutex (lock)
  - Prevents two+ process access the code
  - Supports three operations
    - Lock before running atomic code
    - Unlock after running the code
    - Wait while someone locked the code
- Critical section
  - A code section protected by lock & unlock
- Note
  - Must use the *same* lock for a critical section
  - Must be careful in declaring a critical section
    - What if lock and sleep(10000000000);

```
pthread_mutex_t deposit_lock;

void ProcessRequest(op, accountId, amount) {
  switch (op) {
    case OP_DEPOSIT:
      …
  }
}

void Deposit(accountId, amount) {
  pthread_mutex_lock(&foo_mutex);       // lock before the atomic op.
  account = GetAccount(accountId);
  account->balance += amount;
  StoreAccount(account);
  pthread_mutex_unlock(&foo_mutex);     // unlock after the atomic op.
}

int main(void) {
  int op = -1;
  int accountId, amount = -1, -1;
  pthread_mutex_init(&deposit_lock, NULL);

  while (1) {
    ReceiveRequest(&op, &accountId, &amount);
    ProcessRequest(op, accountId, amount);
  }

  return 0;        // code only reaches here if the server terminates
```

Topics for today

- Part IV – Synchronization
  - Recap:
    - Terminology
    - Process (or thread) scheduling
  - Manage resources
    - Race condition (ATM server's problem)
  - Provide abstraction & Offer standard interface
    - Atomic operation
    - Mutual exclusion (mutex)

Thank You!

Mon/Wed 12:00 – 1:50 PM

Sanghyun Hong

sanghyun.hong@oregonstate.edu

Oregon State University

SAIL
Secure AI Systems Lab