PRINCIPLES OF COMPUTER SYSTEMS DESIGN

CSE130

Winter 2020

CPU Scheduling I

Baskin Engineering UC SANTA CRUZ

Notices

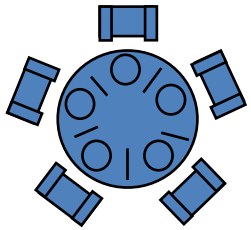
- **Assignment 2** due 23:59 **Sunday February 2**
- **Lab 2** due 23:59 **Sunday February 9**

Today's Lecture

- The Dining Philosophers Problem Revisited
- Introduction to Scheduling
- Scheduling Opportunities
- Scheduling vs. Dispatching
- First Come First Served
- Round Robin
- Assignment 2 Secret Sauce 

The Dining Philosophers Problem

- Five dining philosophers
- Five chopsticks
- Philosophers are either thinking, or eating
- To eat, each philosopher needs two chopsticks
- Chopsticks are put down while thinking



Codifying The Dining Philosophers Problem

- A simple solution is to represent each chopstick by a binary semaphore
- Then use an array of semaphores indexed by position
- A chopstick is picked up with:

  P(chopsticks[pos])
- And set down with:

  V(chopsticks[pos])

Simple Solution - Pseudo Code

```
while (true) {
  P(chopsticks[left]);
  P(chopsticks[right]);
  // nom, nom, nom
  V(chopsticks[left]);
  V(chopsticks[right]);
  // ponder life, the universe, and everything
}
```

If all philosophers pick up their left chopstick at the same time, we get *DEADLOCK* as none of them can now pick up their right chopstick ☹

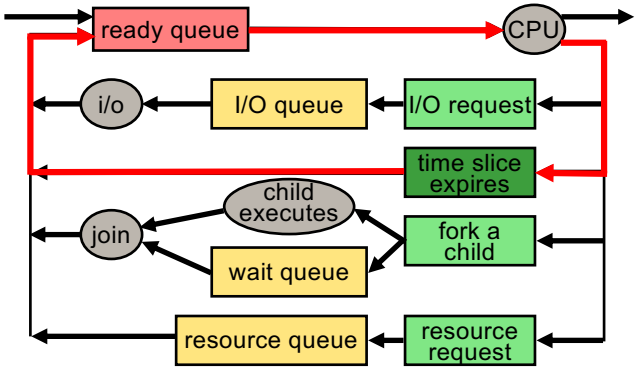

```
#define PHILOSOPHERS    5
#define MIN_EAT_MS    500
#define MAX_EAT_MS   1500
#define MIN_THINK_MS 1000
#define MAX_THINK_MS 3000

// how fast a philosopher can pick up a chopstick,
// 0 = instantaneous
static int muscle_response_time = 0;

typedef struct philosopher_t
{
  int pid;      // philosopher (not process) id
  int noms;     // how many times philosopher has eaten
  sem_t *left;  // binary semaphore for left chopstick
  sem_t *right; // binary semaphore for right chopstick
}
philosopher_t;

static sem_t *chopsticks[PHILOSOPHERS];
static philosopher_t philosophers[PHILOSOPHERS];

static void sleep_ms(int msec) {
  if (msec > 0) {
    struct timespec ts;
    ts.tv_sec = msec / 1000;
    ts.tv_nsec = (msec % 1000) * 1000000;
    nanosleep(&ts, NULL);
  }
}

static void monitor() {
  for (;;) {
    sleep_ms(1000);
    printf("\n");
    for(int i = 0; i < PHILOSOPHERS; i++) {
      printf("  Philosopher %d has eaten %d times\n",
        philosophers[i].pid, philosophers[i].noms);
    }
  }
}
```

```
static void *philosopher_main(void *arg)
{
  philosopher_t *philosopher = (philosopher_t*) arg;

  for (;;) {
    sem_wait(philosopher->left);
    sleep_ms(muscle_response_time);
    sem_wait(philosopher->right);

    sleep_ms(MIN_EAT_MS + rand() % (MAX_EAT_MS - MIN_EAT_MS));

    sem_post(philosopher->left);
    sem_post(philosopher->right);

    philosopher->noms++;
    sleep_ms(MIN_THINK_MS + rand() %
        (MAX_THINK_MS - MIN_THINK_MS));
  }
}


int main(int argc, char* argv[])
{
  muscle_response_time = argc > 1 ? atoi(argv[1]) : 0;
  srand(getpid());
  for (int i = 0; i < PHILOSOPHERS; i++)
    sem_init(&chopsticks[i], 0, 1);

  pthread_t junk;
  for(int i = 0; i < PHILOSOPHERS; i++) {
    philosophers[i].pid = i;
    philosophers[i].right = &chopsticks[i];
    philosophers[i].left = &chopsticks[(i+1) % PHILOSOPHERS];
    pthread_create(
      &junk, NULL, philosopher_main,
      (void*)&philosophers[i]);
  }
  monitor();
}
```


Introduction to CPU Scheduling

- Efficient use of the CPU is critical to overall performance of the entire system
- The CPU is fundamentally a pre-emptible resource
- **Arguably the most important role of the operating system is arbitrating access to the CPU**
- Metrics including priority, previous activity, etc. can be used in determining which process next gets use of the CPU

Preemptive Scheduling (no IO wait)



Scheduling Opportunities

- **Running thread yields** or:
  - Makes a system call: `sleep()`, `fork()` etc.
  - Has to wait for a semaphore, lock, mutex, network socket, etc.
- **Thread terminates**
- **An interrupt occurs**
  - Current thread is still ready
  - Blocked thread(s) become(s) ready
- **Preemption**
  - Usually based on timer interrupt
  - Chose from all ready threads (how you choose is the ***scheduling algorithm***)
- Non-preemption
  - Continue with current thread until it finishes ☹

Scheduling vs. Dispatching

- A **Scheduler**:
  - Maintains the ready queue
  - Selects next process to run from those in the ready queue
- A **Dispatcher**:
  - Puts the selected process onto the CPU by:
    - Switching context (i.e. reloading the PCB)
    - Switching to user mode (i.e jumping to the proper location in the executable code to restart that program where it left off at the previous context switch)
  - **Dispatch Latency**:
    - Time it takes for the dispatcher to stop one process and start another running - an important OS metric
- **NOTE**: The terms "Scheduler" and "Dispatcher" are frequently used interchangeably and inconsistently – try to avoid doing this

Scheduling Criteria

- **Utilization**
  - Try to keep the CPU as busy as possible (used / available)
- **Throughput**
  - # of threads that complete their execution per time unit (threads / time)
- **Turnaround time**
  - Amount of time to execute a particular task start to finish (t_{finish} - t_{start})
- **Waiting time**
  - Amount of time a thread has been waiting in the ready queue (t_{readyQ})
- **Response time**
  - Amount of time from when a request was submitted until the first response is produced ($t_{firstresponse}$ - t_{start})

Scheduling Considerations

- Different schedulers will give different priorities to different goals in different situations
- Throughput depends on the size and type of the jobs
- **Good metrics?**
  - **Batch Systems**: **Turnaround time**
  - **Multiprogrammed**: **Waiting time**
  - **Time Sharing / Interactive**: **Response time**

Waiting Time (for any scheduling algorithm)

For a thread **T** that gets scheduled **N** times during it's lifetime:

arrival = time **T** initially enters the scheduling queue
start$_n$ = time **T** gets the CPU for the **n**th time
finish$_n$ = time **T** yields or is forced to yield the CPU for the **n**th time

$$\text{Waiting Time} = (\text{start}_1 - \text{arrival}) + \sum_{n=2}^{N} (\text{start}_n - \text{finish}_{n-1})$$

First Come First Served (non preemptive)

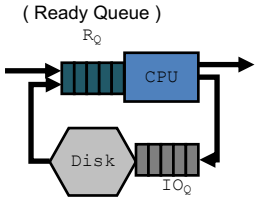
- Non-preemptive FCFS is simple, but...
  - Long running, CPU bound threads introduce highly variable wait times
    - **burst** : time slots threads will use before yielding or having to wait for IO
    - **t_{arrive}** : time slot in which the thread first entered the ready queue

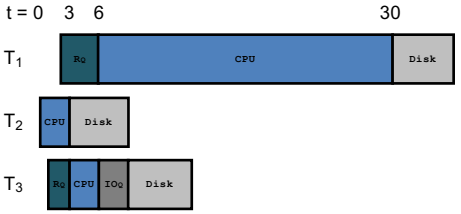
	burst$_1$	t_{arrive}
T_1	24	2
T_2	3	0
T_3	3	1

Waiting times: **T_1** = 6-2 = 4 **T_2** = 0 **T_3** = 3-1 = 2
Mean waiting time: (4 + 0 + 2) / 3 = **2**
Mean turnaround time: (3 + (6-1) + (30-2)) / 3 = **12**
Total turnaround time: **30**

System View of FCFS

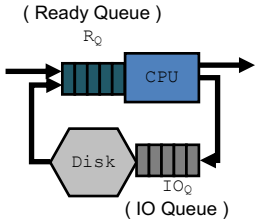
	$burst_1$	t_{arrive}
T_1	24	2
T_2	3	0
T_3	3	1

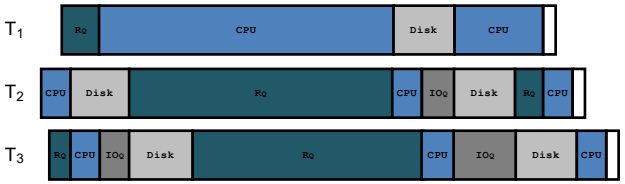




System View of FCFS

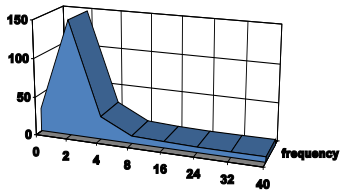
	$burst_1$	$burst_2$	$burst_3$	t_{arrive}
T_1	24	9	-	2
T_2	3	3	3	0
T_3	3	3	3	1





Thread Characterisation

- Multiprogramming aims for maximum CPU utilization
- CPU & I/O Burst Cycles
  - Execution consists of cycles of CPU followed by I/O
  - Both bursts may have to wait in a queue
- CPU burst distribution critical to scheduling algorithm

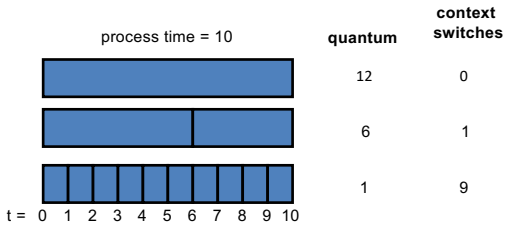


Round Robin

- An attempt to eliminate the convoy effect as they tend to be bad for time-sharing systems
- Each process gets a quantum, **q**, of CPU time
  - Usually $10 \le \mathbf{q} \le 100$ milliseconds
- When quantum expires, process is **preempted** & returned to the ready queue
- If there are **n** processes, each thread gets ~**1/n** of CPU time
- No process waits more than **q∗(n-1)** until its next go on the CPU

Round Robin Performance

- Care must be taken when choosing **q**
  - **q** too large => same as non-preemptive FCFS
  - **q** too small (≅ time to context switch) => overhead is too high



Turnaround Time & Quantum I

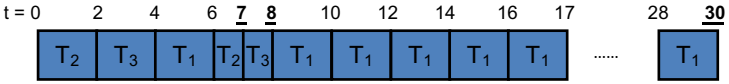
- Turnaround time is impacted by the quantum selected
- With **q** = 2 (assume context switch is ~instantaneous) what is the mean turnaround time?

	$burst_1$	t_{arrive}
T_1	24	2
T_2	3	0
T_3	3	1

Mean turnaround time is:
((7-0) + (8-1) + (30-2)) / 3 = **14**

But that's worse than FCFS ☹

Does it get better if we make **q** = 5?



Turnaround Time & Quantum II

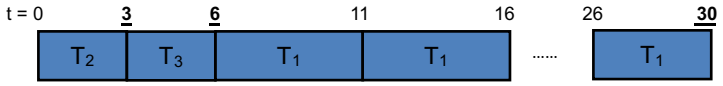
- Can modifying the quantum improve turnaround time?
- With **q** = 5 (assume context switch is ~instantaneous) what is the mean turnaround time?

	$burst_1$	t_{arrive}
T_1	24	2
T_2	3	0
T_3	3	1

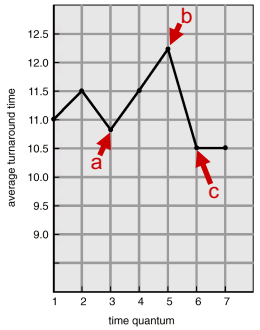
Mean turnaround time is:
((3-0) + (6-1) + (30-2)) / 3 = **12**

But that's still only as good as FCFS 😐

Does it get better if we make **q** = 10?



Turnaround Time & Quantum III



process	time
P_1	6
P_2	3
P_3	1
P_4	7

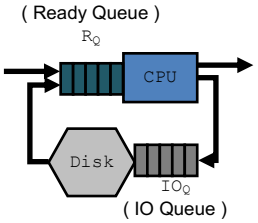
If quantum is ten times the length of a context switch, then 10% of CPU time is spent/wasted context switching

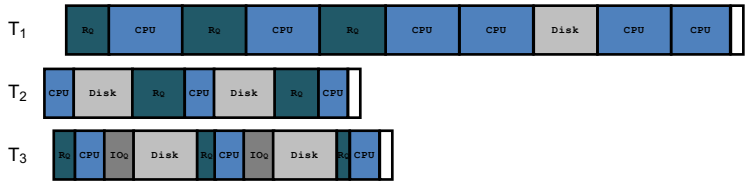
- Turnaround time depends on size of the quantum selected
- Mean turnaround does not always improve with the length of the quantum **(b)**
- Mean turnaround improved (reduced) if most processes (rule of thumb 80%) finish next CPU burst within the next quantum **(a)** and **(c)**

System View of RR

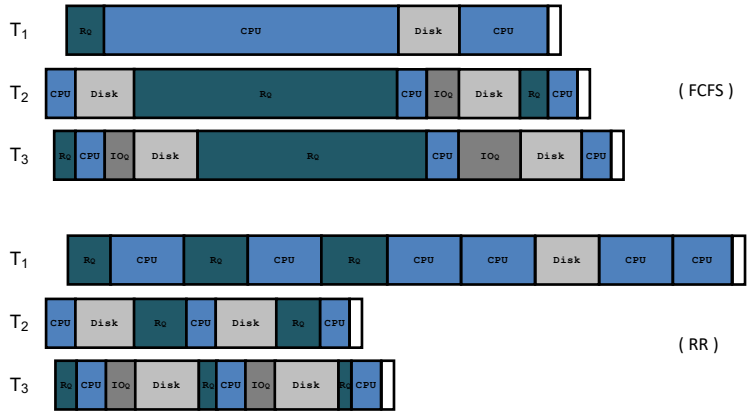
	$burst_1$	$burst_2$	$burst_3$	t_{arrive}
T_1	24	9	-	2
T_2	3	3	3	0
T_3	3	3	3	1

Quantum chosen to exceed most bursts: **q** = 5





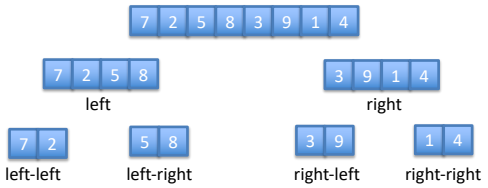
Comparison: FCFS v RR



Conclusion?

- **FCFS**: Bad guys finish first ☹
  - Threads that hog the CPU tend to prevent others from getting a chance to execute
- **RR**: Nice guys finish first ☺
  - Threads that do modest chunks of work before yielding or waiting for IO get more and earlier chances to use the CPU
- In time-sharing situations, RR is usually preferable
- But can we do better than Round Robin?

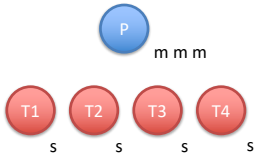
Assignment 2 - Secret Sauce



For 2.6 times speed up, how many threads do we need?
What do these threads do?
Where are the merges done?

Assignment 2 - Process Pseudo Code 1

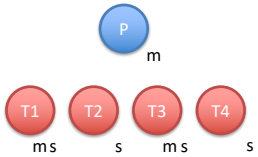
```
split array into four
create four threads to sort a 1/4 of the array each
wait for all four threads to finish
merge left-left & left-right
merge right-left & right-right
merge left and right
```



How may merges are running concurrently?

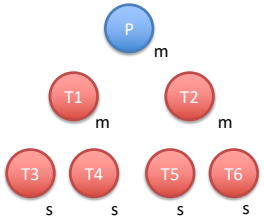
Assignment 2 - Process Pseudo Code 2

```
split array into four
create four threads to sort a 1/4 of the array each
T1 waits for T2 to finish then merges left-left & left-right
T3 waits for T4 to finish then merges right-left & right-right
wait for T1 and T3 to finish
merge left and right
```



Assignment 2 - Process Pseudo Code 3

```
split array into two
create two threads to sort half of the array each
T1 and T2 each create two treads to sort half of their half
T1 waits for T3 and T4 to finish
   then merges left-left & left-right
T2 waits for T5 and T6 to finish
   then merges right-left & right-right
wait for T1 and T2 to finish
merge left and right
```



Next Lecture

- Doing better than Round Robin