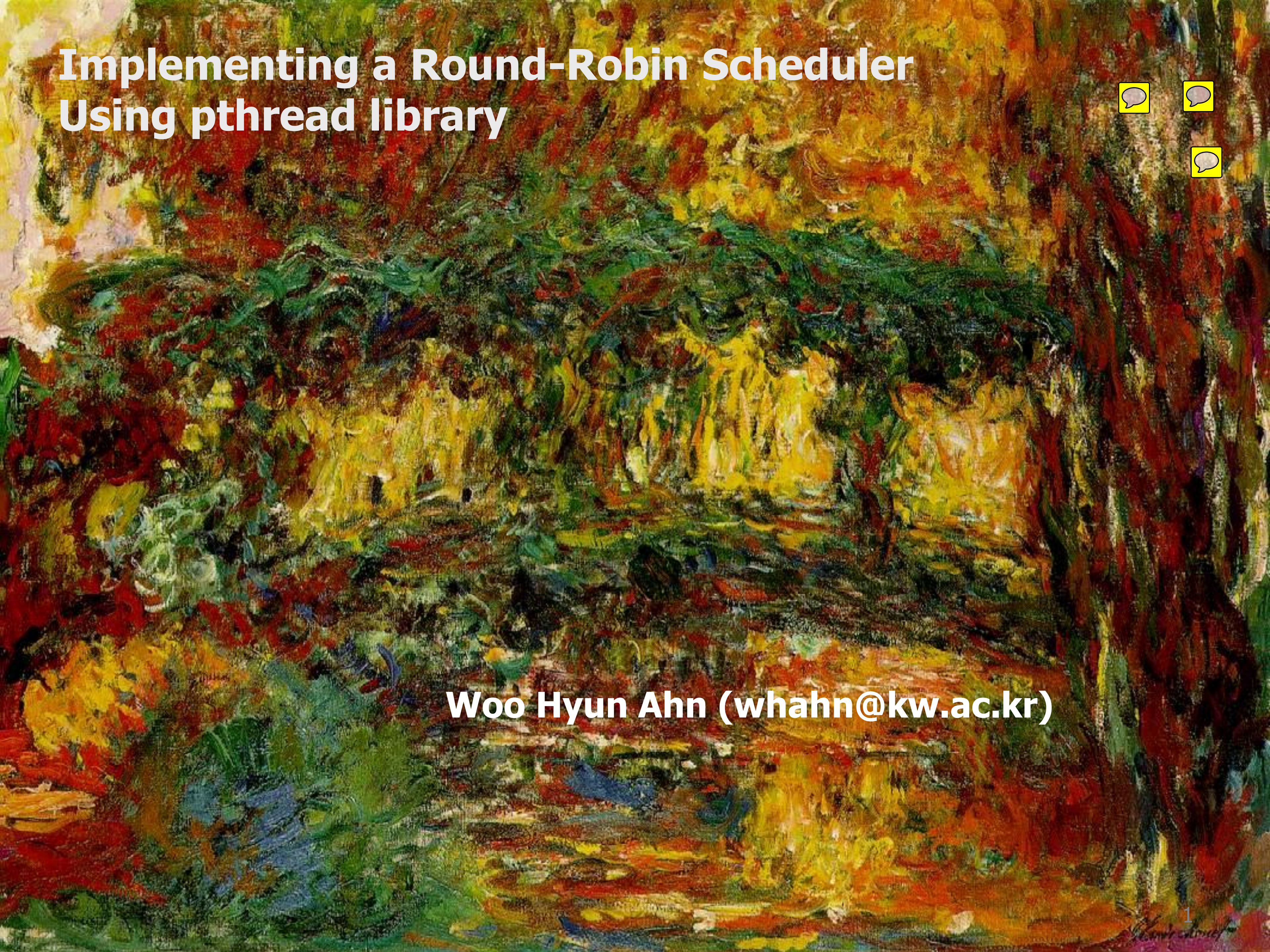
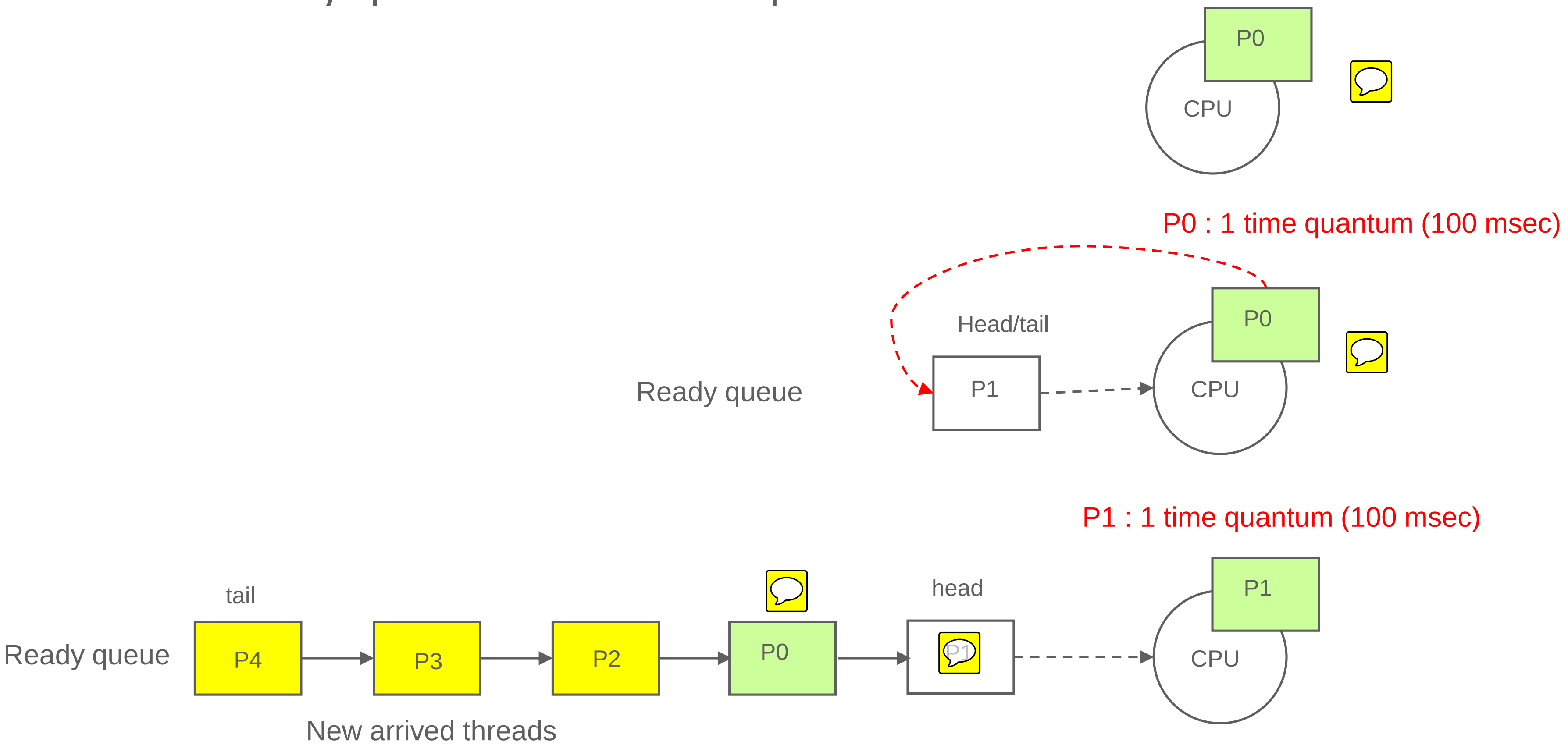


Implementing a Round-Robin Scheduler Using pthread library

Woo Hyun Ahn (whahn@kw.ac.kr)

Round Robin (RR) Scheduling

- Scheduling methodology
  - Each process with the same priority gets a small unit of CPU time (*time quantum or time slice*), usually 10-100 milliseconds. After this time has elapsed, the process is preempted and added to the end of the ready queue. Then the next process in FIFO is executed.



Thread Control Block (TCB)

- Thread Control Block

```
typedef _thread {
    ThreadStatus        status;
    int exitCode;
    pthread_t           tid;
    pthread_cond_t      readyCond;
    BOOL                bReady;
    pthread_mutex_t     readyMutex;
    Thread*             pNext;
    Thread*             pPrev;
} Thread;
```

- Thread status

```
Enum {
        THREAD_STATUS_RUN = 0,
        THREAD_STATUS_READY = 1,
        THREAD_STATUS_SLEEP= 2,
        THREAD_STATUS_ZOMBIE = 3
} ThreadStatus;
```

Thread Control Block (TCB)

- System data structure including thread-specific information.
- TCB contains everything a kernel needs to know about a particular thread.
- Thread status, priority, name, parent/child task information, etc.

Creating a thread with thread

A thread is created with

```
int thread_create(
    thread_t *thread,
    const thread_attr_t *attr,
    void *(*start_routine)(void *),
    void *arg);
```

- The creating thread must provide a location for storage of the thread id.

- The third parameter is just the name of the function for the thread to run.

- The last parameter is a pointer to the arguments.

The Thread ID

thread_t thread_self(void)

- Each thread has an id of type thread_t.
  - On most systems this is just an integer (like a process ID)
  - But it does not have to be

thread_join and thread_exit

```
int thread_join(thread_t thread, void** retval);
```

- thread_join() is a blocking call on threads
- It indicates that the caller wishes to block until the thread being joined exits.

- `int thread_exit(void* retval);`
  - Should be called before thread is terminated.

Thread_suspend(thread_t tid)

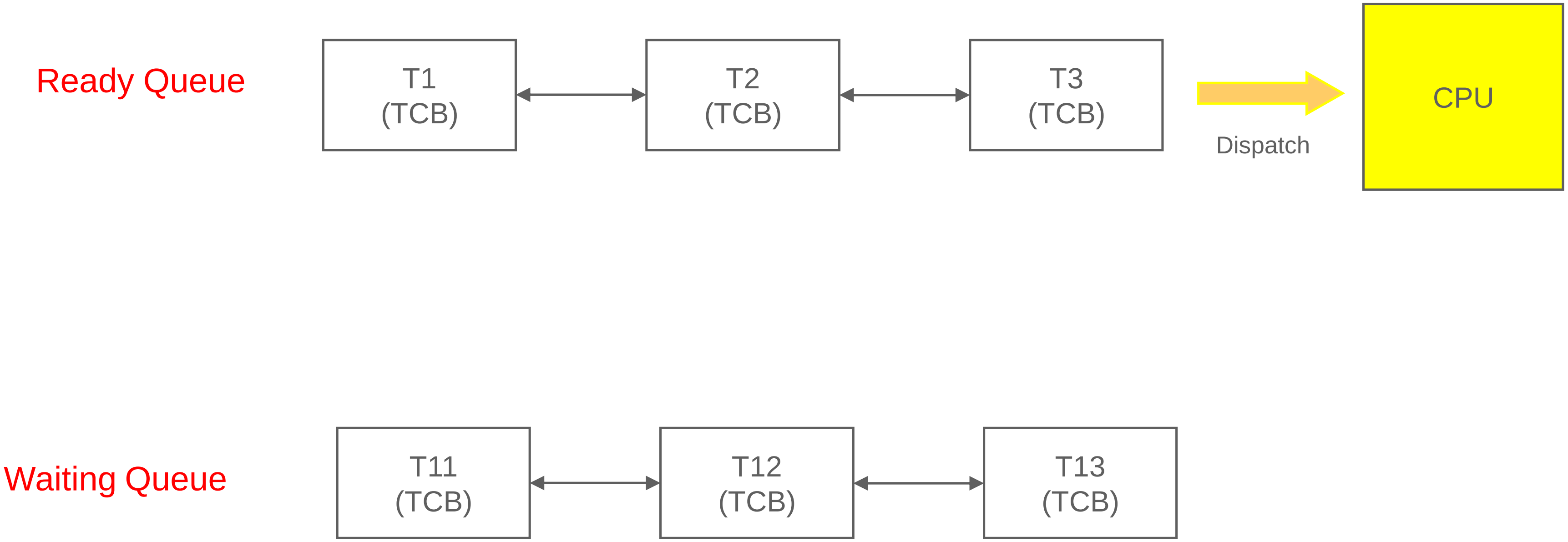
- Suspends a thread.
- SYNOPSIS
  - **int thread_suspend(thread_t tid);**
- **Parameters**
  - *tid*
    - [in] thread ID of a thread to suspend.
- **Return Values**
  - If the function succeeds, the return value is 0; otherwise, it is (-1).

Thread_resume(thread_t tid)

- Resume a specific suspended task.
- **SYNOPSIS**
  - **int thread_resume(thread_t tid);**
- **Parameters**
  - *tid*
    - [in] thread ID of a thread to resume.

- **Return Values**
  - If the function succeeds, the return value is 0; otherwise, it is (-1).

Ready Queue & Task Waiting Queue



Ready Queue

T1 (TCB)

T2 (TCB)

T3 (TCB)

Dispatch

CPU

Waiting Queue

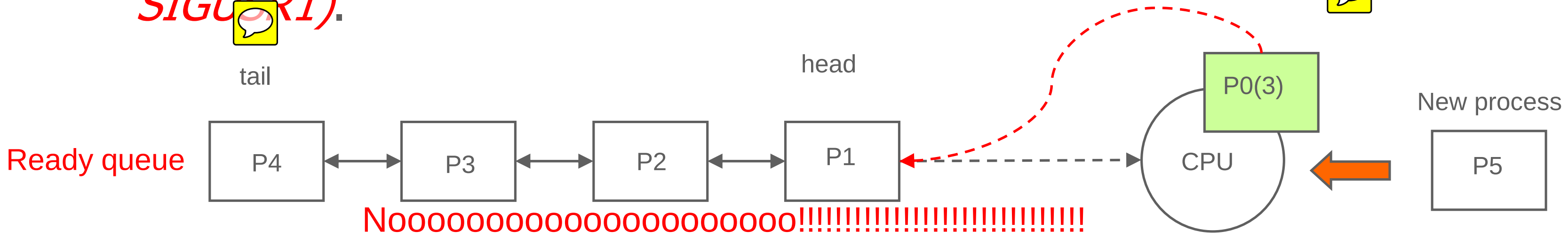
T11 (TCB)

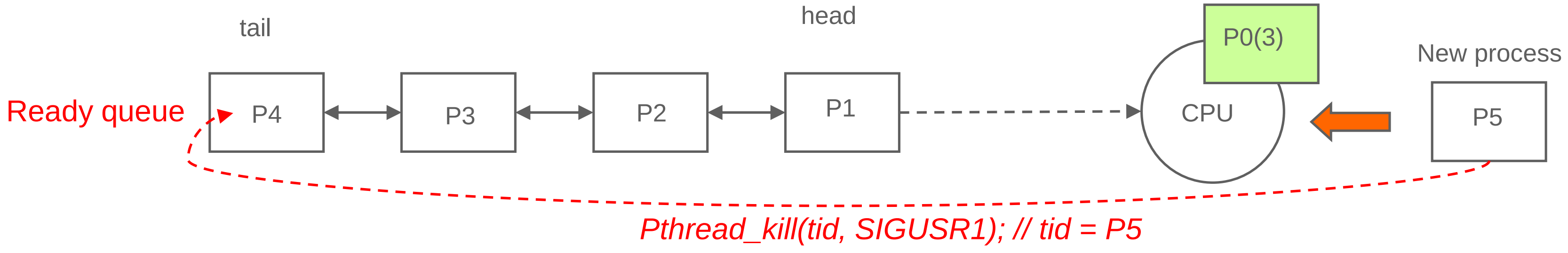
T12 (TCB)

T13 (TCB)

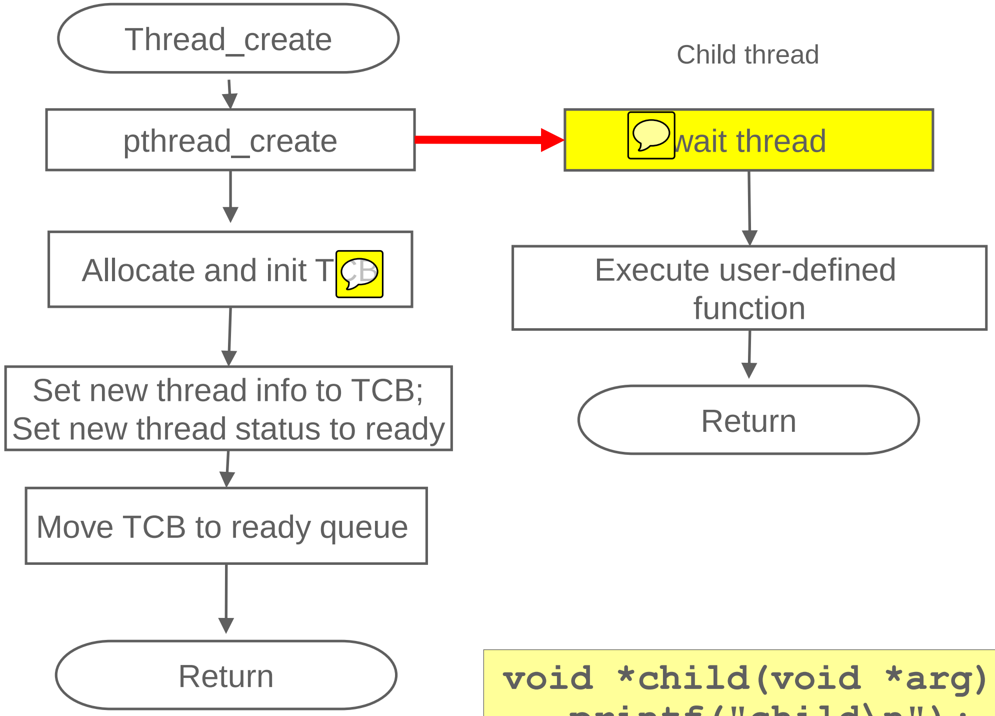
Ready Queue & Sleep Queue (Cont'd)

- A newly created thread is not run soon, but first moved at the tail of the ready queue.
  - Threads in the ready queue should be executed in the round-robin manner.
  - The created thread should be waited until the scheduler sequentially execute other threads that are placed in the ready queue.
  - When a thread is created, the thread is waited by *__thread_wait_handler*, and the thread is executed by *pthread_kill (tid, SIGUSR1)*.





thread_create



Thread_create

pthread_create

Allocate and init TCB

Set new thread info to TCB;
Set new thread status to ready

Move TCB to ready queue

Return

Child thread

wait thread

Execute user-defined function

Return

- Child thread is suspended when it is created.
  - Otherwise, the scheduler cannot control the child thread.
- How can the child execute the waiting code?
  - Use a wrapper function that includes the user-defined function when calling pthread_create function.

```
void *child(void *arg) {
   printf("child\n");
   return NULL;
}
int main(int argc, char *argv[])
{
   pthread_t c;
   pthread_create(&c, NULL, child, NULL);
   pthread_join(c);
   return 0;
}
```

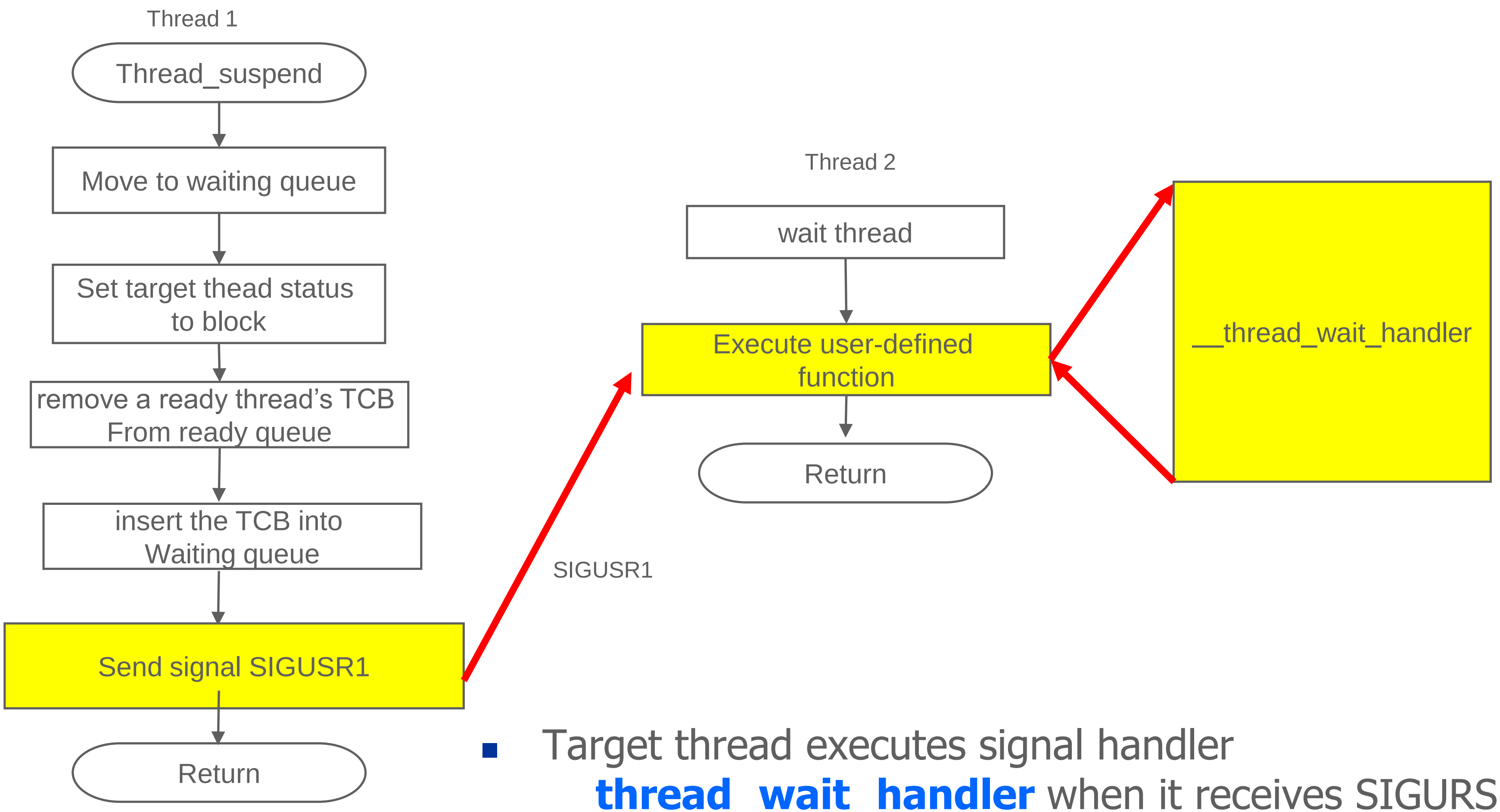
Wrapper Function

- Thread_create creates a child calling the wrapper function.

```
void *child(void *arg) {
   printf("child\n");
   return NULL;
}


void* __wrapperFunc(void* arg)
{
  void* ret;
  WrapperArg* pArg = (WrapperArg*)arg;

  void* funcPtr = pArg.funcptr;
  void* funcArg = pArg.funcArg;

  ret = (*funcPtr)(funcArg);
  return ret;
}
```

```
Typedef __wrapperArg {
   void  (*funcPtr)(void*);
   void* funcArg;
} WrapperArg;


int main(int argc, char *argv[])
{
   pthread_t c;
   WrapperArg wrapperArg;
   wrapperArg.funcptr = child;
   wrapper.funcArg = 10;
   Pthread_create(&c, NULL,
     __wrapperFunc,&wrapperArg);
   pthread_join(&c);
   return 0;
}
```

thread_suspend


Thread 1
Thread_suspend
Move to waiting queue
Set target thead status to block
remove a ready thread's TCB From ready queue
insert the TCB into Waiting queue
Send signal SIGUSR1
Return
SIGUSR1
Thread 2
wait thread
Execute user-defined function
Return
__thread_wait_handler

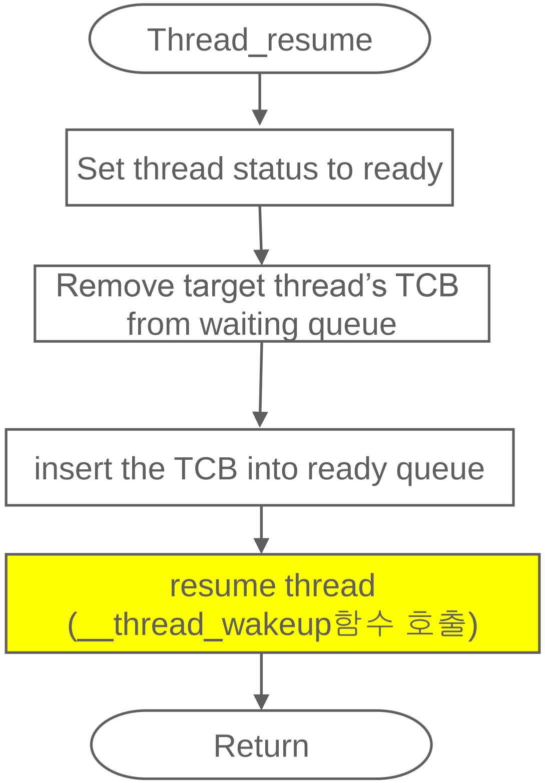
- Target thread executes signal handler **__thread_wait_handler** when it receives SIGURS1
- A thread is suspend in a signal handler **__thread_wait_handler** when it receives signal SIGUSR1

Thread Wait using Signal Handler

```
void __thread_wait_handler(int signo)
{
   Thread* pTh;

   pTh = __getThread(pthread_self());
   pthread_mutex_lock(&(pTh->readyMutex));
   while (pTh->bRunnable == FALSE)
      pthread_cond_wait(&(pTh->readyCond), &(pTh->readyMutex));
   pthread_mutex_unlock(&(pTh->readyMutex));
}
```

- Signal handler for thread waiting
  - Thread_suspend sends signal SIGUSR1 to the target thread.
  - __thread_wait_handler is invoked to sleep the target thread.

thread_create, thread_suspend, thread_resume


Thread_resume
Set thread status to ready
Remove target thread’s TCB from waiting queue
insert the TCB into ready queue
resume thread (__thread_wakeup함수 호출)
Return

__thread_wakeup

- __thread_wakeup
  - Signals on the condition variable to wake up the target thread that is blocked on the condition variable in __thread_wait_handler()

```
void __thread_wakeup(Thread* pTh)
{
   pthread_mutex_lock(&(pTh->readyMutex));
   pTh->bRunable = TRUE;
   pthread_cond_signal(&(pTh->readyCond));
   pthread_mutex_unlock(&(pTh->readyMutex));
}
```

Thread join

Example 1

```
int done = 0;

void *child(void *arg) {
   printf("child\n");
   done = 1;
   return NULL;
}

int main(int argc, char *argv[])
{
   pthread_t c;
   printf("parent: begin\n");
   pthread_create(&c, NULL,
      child, NULL);
   while (done == 0); // spin
   printf("parent: end\n");
   return 0;
}
```

Example 2

```
int done = 0;
pthread_mutex_t m =
    PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t c =
    PTHREAD_COND_INITIALIZER;
void *child(void *arg) {
   printf("child\n");
   pthread_mutex_lock(&m);
   done = 1;
   pthread_cond_signal(&c);
   pthread_mutex_unlock(&m);
   return NULL;
}
int main(int argc, char *argv[])
{
   pthread_t c;
   printf("parent: begin\n");
   pthread_mutex_lock(&m);
   while (done == 0)
      pthread_cond_wait(&c, &m);
   pthread_mutex_unlock(&m);
   printf("parent: end\n");
   return 0;
}
```

Example: Thread join

Example 2

```
int done = 0;
pthread_mutex_t m =
    PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t c =
    PTHREAD_COND_INITIALIZER;
void *child(void *arg) {
   printf("child\n");
   pthread_mutex_lock(&m);
   done = 1;
   pthread_cond_signal(&c);
   pthread_mutex_unlock(&m);
   return NULL;
}
int main(int argc, char *argv[])
{
   pthread_t c;
   printf("parent: begin\n");
   pthread_mutex_lock(&m);
   while (done == 0)
      pthread_cond_wait(&c, &m);
   pthread_mutex_unlock(&m);
   printf("parent: end\n");
   return 0;
}
```

mythread_exit()

mythread_join()

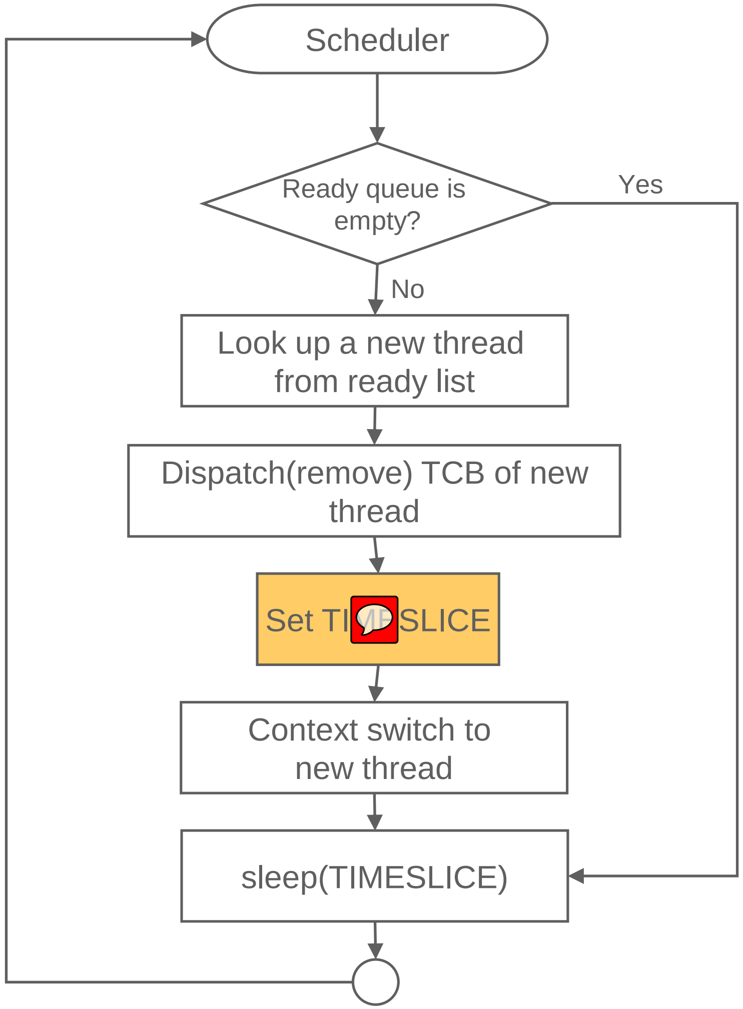
Example 3

```
int done = 0;
pthread_mutex_t m =
    PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t c =
    PTHREAD_COND_INITIALIZER;
void *child(void *arg) {
   printf("child\n");
   mythread_exit();
   return NULL;
}
int main(int argc, char *argv[])
{
   pthread_t c;
   printf("parent: begin\n");
   mythread_join();
   printf("parent: end\n");
   return 0;
}
```

Thread Scheduler

- Task scheduler is a background thread that is not seen to applications.
  - The initialization thread (or main thread) is the scheduler thread
- Time slice
  - #define TIMESLICE (2)
- `void RunScheduler(void)`
  - Run thread scheduler.
  - Implemented by sleep()

Task Scheduler



Scheduler
Ready queue is empty?
Yes
No
Look up a new thread from ready list
Dispatch(remove) TCB of new thread
Set TIMESLICE
Context switch to new thread
sleep(TIMESLICE)

Context Switching

- Procedures
  - Stop the current running thread: *pthread_kill(curtid, SIGUSR1)*
  - Execute a given target thread that is intended to be executed (i.e., scheduled, or dispatched) in the next order: *__thread_wakeup(pNewThread)*
  - pNewThread can be obtained by tid of the target thread to be executed.

- Interface
  - void _ContextSwitch(Thread pCurThread, Thread* pNewThread)

System Initialization

- System initialization routine creates the following:
  - Scheduling queues (ready queue and waiting queue)
  - Thread scheduler.

Main.c

```
void main()
{
     thread_id tid;
     int arg;
     Init();

     thread_create(&tid, NULL, AppTask, &arg);

    RunScheduler();

}
```

Init.c

```
void Init(void)
{

    // Create ready queue and waiting queue
    …
    // initialize thread scheduler
}
```

App.c

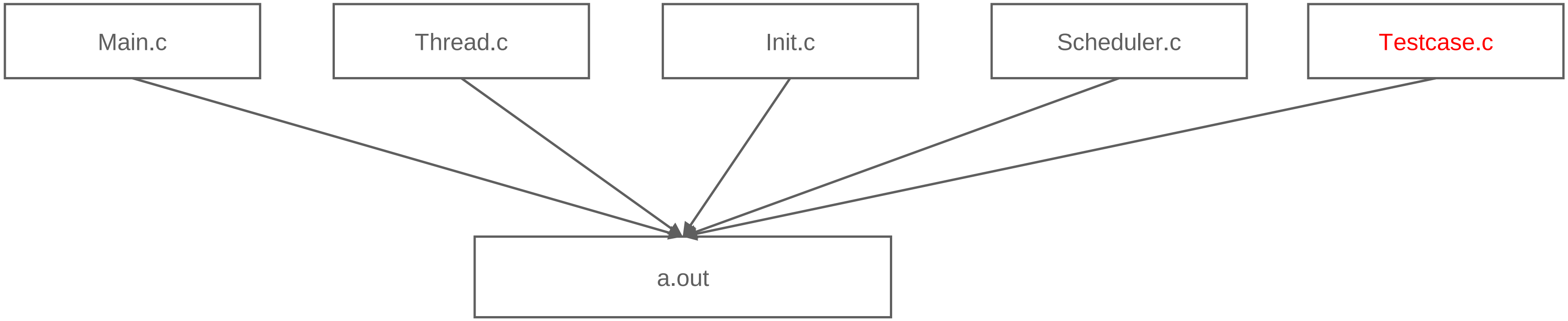
```
Void* AppTask(void* param)
{
     TestCase();
     return NULL;
}
```

Testcase

Testcase.c

```
void* foo1(void* arg)
{
    ….
    while(1);
}

void foo2(void* arg)
{
 ….
   while(1);
}
…

void Testcase(void)
{
     int tid1, tid2, tid3;
     int  arg1, arg2, arg3;

     thread_create(&tid1, NULL, foo1, &arg1);
     thread_create(&tid2, NULL, foo2, &arg2);
     thread_create(&tid3, NULL, foo3, &arg3);
     …
     thread_suspend(tid1);
     thread_suspend(tid2);
     …
     thread_resume(tid1);
     while(1);
}
```

Building



Main.c
Thread.c
Init.c
Scheduler.c
Testcase.c
a.out